

Abstract Learning Frameworks for Synthesis

Christof Löding¹, P. Madhusudan², and Daniel Neider^{2,3}

¹ RWTH Aachen

² University of Illinois, Urbana-Champaign

³ University of California, Los Angeles

Abstract. We develop abstract learning frameworks for synthesis that embody the principles of the CEGIS (counterexample-guided inductive synthesis) algorithms in current literature. Our framework is based on iterative learning from a hypothesis space that captures synthesized objects, using counterexamples from an abstract sample space, and a concept space that abstractly defines the semantics of synthesis. We show that a variety of synthesis algorithms in current literature can be embedded in this general framework. We also exhibit three general recipes for convergent synthesis: the first two recipes based on finite spaces and Occam learners generalize all techniques of convergence used in existing engines, while the third, involving well-founded quasi-orderings, is new, and we instantiate it to concrete synthesis problems.

1 Introduction

The field of synthesis, which includes several forms of synthesis including synthesizing controllers [44], program expressions [51], program repairs [32], program translations [15,29], loop invariants [22,21], and even entire programs [37,31], has become a fundamental and vibrant subfield in programming languages. While classical studies of synthesis have focused on synthesizing entire programs or controllers from specifications [37,44], there is a surge of tractable methods that have emerged in recent years in synthesizing small program expressions. These expressions often are complex but small, and are applicable in niche domains such as program sketching [51] (finding program expressions that complete code), synthesizing Excel programs for string transformations [23], synthesizing super-optimized code [48], deobfuscating code [27], synthesizing invariants to help in verification [21,22], etc.

One prominent technique that has emerged in recent years for expression synthesis is based on *inductively learning expressions from samples*. Assume the synthesis problem is to synthesize an expression e that satisfies some specification $\psi(e)$. The crux of this approach is to *ignore* the precise specification ψ , and instead synthesize an expression based on certain *facets* of the specification. These incomplete facets of the specification are often much simpler in structure and in logical complexity compared to the specification, and hence synthesizing an expression satisfying the constraints the facets impose is more tractable. The learning-based approach to synthesis hence happens in rounds— in each round,

the learner synthesizes an expression that satisfies the current facets, and a *verification oracle* checks whether the expression satisfies the actual specification ψ , and if not, finds a new facet of the specification witnessing this. The learner then continues to synthesize by adding this new facet to its collection.

This counter-example guided inductive synthesis (CEGIS) approach [50] to synthesis in current literature philosophically advocates precisely this kind of inductive synthesis. The CEGIS approach has emerged as a powerful technique in several domains of both program synthesis as well as program verification ranging from synthesizing program invariants for verification [21,22] to specification mining [3], program expressions that complete sketches [51], superoptimization [48], control [28], string transformers for spreadsheets [23], protocols [54], etc.

The goal of this paper is to develop a *theory of iterative learning-based synthesis* through a formalism we call *abstract learning frameworks for synthesis*. The framework we develop aims to be general and abstract, encompassing several known CEGIS frameworks as well as several other synthesis algorithms not generally viewed as CEGIS. The goal of this line of work is to build a framework, with accompanying concepts, definitions, and vocabulary that can be used to understand and combine learning-based synthesis across different domains.

An abstract learning framework (ALF) (see Figure 1 on Page 5) consists of three spaces: \mathcal{H} , \mathcal{S} , and \mathcal{C} . The (*semantic*) *concept space* \mathcal{C} gives semantic descriptions of the concepts that we wish to synthesize, the *hypotheses space* \mathcal{H} comprises restricted (typically syntactically restricted) forms of the concepts to synthesize, and the sample space \mathcal{S} consists of samples (modeling facets of the specification) from which the learner synthesizes hypotheses. The spaces \mathcal{H} and \mathcal{S} are related by a variety of functions that give semantics to samples and semantics to hypotheses using the space \mathcal{C} . The conditions imposed on these relations capture the learning problem precisely, and their abstract formulation facilitates modeling a variety of synthesis frameworks in the literature.

The target for synthesis is specified as a *set* of semantic concepts. This is an important digression from classical learning frameworks, where often one can assume that there is a *particular* target concept that the learner is trying to learn. Note that in synthesis problems, we must *implement the teacher as well*, and hence the modeling of the target space is important. In synthesis problems, the teacher does not have a single target in mind nor does she know explicitly the target set (if she knew, there would be no reason to synthesize!). Rather, she knows the *properties* that capture the set of target concepts. For instance, in invariant synthesis, the teacher knows the properties of a set being an invariant for a loop, and this defines implicitly a *set* of invariants as target. The teacher needs to examine a hypothesis and check whether it satisfies the properties defining the target set. Consequently, we can view the teacher as a *verification oracle* that checks whether a hypothesis belongs to the implicitly defined target set.

We exhibit a variety of existing synthesis frameworks that can be naturally seen as instantiations of our abstract-learning framework, where the formulation shows the diversity in the instantiations of the spaces. These include (a) a variety of CEGIS-based synthesis techniques for synthesizing program expressions in

sketches (completing program sketches [51], synthesizing loop-free programs [24], mining specifications [28], synthesizing synchronization code for concurrent programs [13], etc.), (b) synthesis from input-output examples such as Flashfill [23], (c) the CEGIS framework applied to the concrete problem of solving synthesis problems expressed in the SMT-based SyGuS format [2,1], and three synthesis engines that use learning to synthesize solutions, (d) invariant synthesis frameworks, including Houdini [18] and the more recent ICE-learning model for synthesizing loop invariants [21], spanning a variety of domains from arithmetic [21,22] to quantified invariants over data structures [20], and (e) synthesizing fixed-points and abstract transformers in abstract interpretation settings [52].

Formalizing of synthesis algorithms as ALFs can help highlight the nuances of different learning-based synthesis algorithms, even for the *same* problem. One example comprises two inductive learning approaches for synthesizing program invariants— one based on the ICE learning model [21], and the second which is any synthesis engine for logically specified synthesis problems in the SyGuS format, which can express invariant synthesis. Though both can be seen as CEGIS-based synthesis algorithms, the sample space for them are very different, and hence the synthesis algorithms are also different— the significant performance differences between SyGuS-based solvers and ICE-based solvers (the latter performing better) in the recent SyGuS competition (invariant-synthesis track) suggest that this choice may be crucial [4]. Another example are two classes of CEGIS-based solvers for synthesizing linear integer arithmetic functions against SyGuS specifications— one based on a sample space that involves purely inputs to the function being synthesized [46,41], while the other is the more standard CEGIS algorithm based on valuations of quantified variables.

We believe that just describing an approach as a learning-based synthesis algorithm or a CEGIS algorithm does not convey the nuances of the approach— it is important to precisely spell out the sample space and the semantics of this space with respect to the space of hypotheses being learned. The ALF framework gives the vocabulary in phrasing these nuances, allowing us to compare and contrast different approaches.

Convergence: The second main contribution of this paper is to study *convergence* issues in the general abstract learning-based framework for synthesis. We first show that under the reasonable assumptions that the learner is consistent (always proposes a hypothesis consistent with the samples it has received) and the teacher is honest (gives a sample that distinguishes the current hypothesis from the target set without ruling out any of the target concepts), the iterative learning will always converge *in the limit* (though, not necessarily in finite time, of course). This theorem vouches for the correctness of our abstract formalism in capturing abstract learning, and utilizes all the properties that define ALFs.

We then turn to studying strategies for convergence in finite time. We propose three general techniques for ensuring successful termination for the learner. First, when the hypothesis space is bounded, it is easy to show that any consistent learner (paired with an honest teacher) will converge in finite time. Several examples of these exist in learning— learning conjunctions as in the Houdini algorithm [18],

etc., learning Boolean functions (like decision-tree learning with purely Boolean predicates as attributes) or functions over bit-vector domains (Sketch [51] and the SyGuS solvers that work on bit-vectors), and learning invariants using specialized forms of a finite class of automata that capture list/array invariants [20].

The second recipe is a formulation of the Occam’s razor principle that uses parsimony/simplicity as the learning bias [7]. The idea of using Occam’s principle in learning is prevalent (see Chapter 2 of [30] and [38]) though its universal appeal in generalizing concepts is debatable [17]. We show, however, that learning using Occam’s principle helps in convergence. A learner is said to be an Occam learner if there is a *complexity ordering*, which needs to be a total quasi order where the set of elements below any element is finite, such that the learner always learns a *smallest* concept according to this order that is consistent with the sample. We can then show that any Occam learner will converge to some target concept, if one exists, in finite time. This result generalizes many convergent learning mechanisms that we know of in the literature (for example, the convergent ICE-learning algorithms for synthesizing invariants using constraint solvers [21], and the enumerative solvers in almost every domain of synthesis [34,32,42,54], including for SyGuS [2,1], that enumerate by dovetailing through expressions).

The first two recipes for finite convergence cover all the methods we know in the literature for convergent learning-based synthesis, to the best of our knowledge. The third recipe for finite convergence is a more complex one based on well-founded quasi orderings. This recipe is involved and calls for using clever initial queries that force the teacher to divulge information that then makes the learning space tractable. We do not know of any existing synthesis learning frameworks that use this natural recipe, but propose two new convergent learning algorithms following this recipe, one for intervals, and the other for conjunctive linear inequality constraints over a set of numerical attributes over integers.

2 Abstract Learning Frameworks for Synthesis

In this section we introduce our abstract learning framework for synthesis. Figure 1 gives an overview of the components and their relations that are introduced in the following (ignore the target \mathcal{T} , $\gamma^{-1}(\mathcal{T})$, and the maps τ and λ for now). We explain these components in more detail after the formal definition.

Definition 1 (Abstract Learning Frameworks). *An abstract learning framework for synthesis (ALF, for short), is a tuple $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s), \gamma, \kappa)$, with*

- *A class \mathcal{C} , called the concept space,*
- *A class \mathcal{H} , called the hypothesis space,*
- *A class \mathcal{S} , called the sample space, with a join semi-lattice $(\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s)$ defined over it,*
- *A concretization function $\gamma : \mathcal{H} \rightarrow \mathcal{C}$, and*
- *A consistency function: $\kappa : \mathcal{S} \rightarrow 2^{\mathcal{C}}$ satisfying $\kappa(\perp_s) = \mathcal{C}$ and $\kappa(S_1 \sqcup S_2) = \kappa(S_1) \cap \kappa(S_2)$ for all $S_1, S_2 \in \mathcal{S}$. If the second condition is relaxed to $\kappa(S_1 \sqcup S_2) \subseteq \kappa(S_1) \cap \kappa(S_2)$, we speak of a general ALF.*

We say an ALF has a complete sample space if the sample space $(\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s)$ is a complete join semi-lattice (i.e., if the join is defined for arbitrary subsets of \mathcal{S}). In this case, the consistency relation has to satisfy $\kappa(\sqcup(S')) = \bigcap_{S \in S'} \kappa(S)$ for each $S' \subseteq \mathcal{S}$ (and $\kappa(\sqcup(S')) \subseteq \bigcap_{S \in S'} \kappa(S)$ for general ALFs).

As in computational learning theory, as presented in e.g., in [6] or [30], we consider a *concept space* \mathcal{C} , which contains the objects that we are interested in. For example, in an invariant synthesis setting in verification, an element $C \in \mathcal{C}$ would be a *set* of program configurations. In the synthesis setting, \mathcal{C} could contain the objects we would like to synthesize, such as all functions from \mathbb{Z}^n to \mathbb{Z} .

The *hypothesis space* \mathcal{H} contains the objects that the learner produces. These are representations of (some) elements from the concept space. For example, if \mathcal{C} consists of all functions from \mathbb{Z}^n to \mathbb{Z} , then \mathcal{H} could consist the set of all functions expressible in linear arithmetic.

The relation between hypotheses and concepts is given by a concretization function $\gamma : \mathcal{H} \rightarrow \mathcal{C}$ that maps hypotheses to concepts (their semantics).

In classical computational learning theory for classification [30,38], one often considers samples consisting of positive and negative examples. If learning is used to infer a target concept that is not uniquely defined but rather should satisfy certain properties, then samples consisting of positive and negative examples are sometimes not sufficient. As we will show later, samples can be quite complex (see Section 4 for such examples, including implication counterexamples and grounded formulas).

We work with a *sample space*, which is a bounded join-semilattice $(\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s)$ (i.e., \sqsubseteq_s is a partial order over \mathcal{S} with \perp_s as the least element, and \sqcup is the binary least upper-bound operator on \mathcal{S} with respect to this ordering). An element $S \in \mathcal{S}$, when given by the teacher, intuitively, gives some information about a target specification. The join is used by the learner to combine the samples returned as feedback by the teacher during iterative learning. The least element \perp_s corresponds to the empty sample. We encourage the reader to think of the join as the union of samples.

The *consistency relation* κ captures the semantics of samples with respect to the concept space by assigning to each sample S the set $\kappa(S)$ of concepts that are consistent with the sample. The first condition on κ says that all concepts are consistent with the empty sample \perp_s . The second condition says that the set of samples consistent with the join of two samples is precisely the set of concepts that is consistent with both the samples. Intuitively, this means that joining samples does not introduce new inconsistencies, and existing inconsistencies transfer to bigger samples. The condition that $\kappa(S_1 \sqcup S_2) \subseteq \kappa(S_1) \cap \kappa(S_2)$ is natural, as

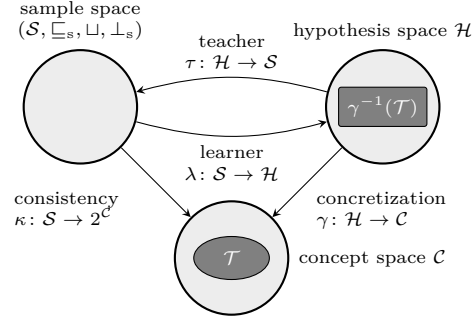


Fig. 1. Components of an ALF

it says that if a concept is consistent with the join of two samples, then the concept must be consistent with both of them individually. The condition that $\kappa(S_1 \sqcup S_2) \supseteq \kappa(S_1) \cap \kappa(S_2)$ is debatable; it claims that samples when taken together cannot eliminate a concept that they couldn't eliminate individually. We therefore mention the notion of *general ALF* in Definition 1. However, we have not found any natural example that requires such a generalization, and therefore prefer to work with ALFs instead of general ALFs in the rest of the paper. In Definition 4, we comment on what needs to be adapted to make the results of the paper go through for general ALFs.

The following simple but useful observation on monotonicity of the consistency relation (which easily follows from the property of the consistency relation, for ALFs and general ALFs) is used in some of the proofs.

Remark 1. If $S_1 \sqsubseteq_s S_2$, then $\kappa(S_2) \subseteq \kappa(S_1)$.

Some other auxiliary definitions we will need: We define $\kappa_{\mathcal{H}}(S) := \{H \in \mathcal{H} \mid \gamma(H) \in \kappa(S)\}$ to be the set of hypotheses that are consistent with S . For a sample $S \in \mathcal{S}$ we say that S is *realizable* if there exists a hypothesis that is consistent with S (i.e., $\kappa_{\mathcal{H}}(S) \neq \emptyset$).

ALF Instances and Learners An instance of a learning task for an ALF is given by a specification that defines target concepts. The goal is to infer a hypothesis whose semantics is such a target concept. In classical computational learning theory, this target is a unique concept. In applications for synthesis, however, there can be many possible target concepts, for example, all inductive invariants of a program loop.

Formally, a *target specification* is just a set $\mathcal{T} \subseteq \mathcal{C}$ of concepts. An ALF instance combines an ALF and a target specification:

Definition 2 (ALF Instance). An ALF instance is a pair (A, \mathcal{T}) where $A = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s), \gamma, \kappa)$ is an ALF and $\mathcal{T} \subseteq \mathcal{C}$ is a target specification.

The goal of learning-based synthesis is for the learner to synthesize *some* element $H \in \mathcal{H}$ such that $\gamma(H) \in \mathcal{T}$. Furthermore, the role of the teacher is to instruct the learner giving reasons why the hypothesis produced by the learner in the current round does not belong to the target set.

There is a subtle point here worth emphasizing. In synthesis frameworks, the teacher does not explicitly know the target space \mathcal{T} . Rather she knows a definition of the target space, and she can examine a hypothesis H and check whether it satisfies the properties required of the target set. For instance, when synthesizing an invariant for a program, the teacher knows the properties of the invariant (inductiveness, etc.) and gives counterexample samples based on failed properties.

We say that the target specification is *realizable by a hypothesis*, or simply *realizable*, if there is some $H \in \mathcal{H}$ with $\gamma(H) \in \mathcal{T}$. For a hypothesis $H \in \mathcal{H}$, we often write $H \in \mathcal{T}$ instead of $\gamma(H) \in \mathcal{T}$.

As in classical computational learning theory, we define a *learner* (see Figure 1) to be a function that maps samples to hypotheses, and a consistent learner to be a learner that only proposes consistent hypotheses for samples.

Definition 3. A learner for an ALF $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s), \gamma, \kappa)$ is a map $\lambda : \mathcal{S} \rightarrow \mathcal{H}$ that assigns a hypothesis to every sample. A consistent learner is a learner λ with $\gamma(\lambda(S)) \in \kappa(S)$ for all realizable samples $S \in \mathcal{S}$.

Iterative learning. In the iterative learning setting, the learner produces a hypothesis starting from some initial sample (e.g., \perp_s). For each hypothesis provided by the learner that does not satisfy the target specification, a teacher (see Figure 1) provides feedback by returning a sample witnessing that the hypothesis does not satisfy the target specification.

Definition 4. Let (A, \mathcal{T}) be an ALF instance with $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s), \gamma, \kappa)$, and $\mathcal{T} \subseteq \mathcal{C}$. A teacher for this ALF instance is a function $\tau : \mathcal{H} \rightarrow \mathcal{S}$ that satisfies the following two properties:

- i) **Progress:** $\tau(H) = \perp_s$ for each target element $H \in \mathcal{T}$, and $\gamma(H) \notin \kappa(\tau(H))$ for all $H \notin \mathcal{T}$, and
- ii) **Honesty:** $\mathcal{T} \subseteq \kappa(\tau(H))$ for each $H \in \mathcal{H}$.⁴

Firstly, progress says that if the hypothesis is in the target set, then the teacher must return the “empty” sample \perp_s , signaling that the learner has learned a target; otherwise, the teacher must return a sample that rules out the current hypothesis. This ensures that a consistent learner can never propose the same hypothesis again, and hence makes progress. Secondly, honesty demands that the sample returned by the teacher is consistent with *all* target concepts. This ensures that the teacher does not eliminate any element of the target set arbitrarily.

When the learner and teacher interact iteratively, the learner produces a sequence of hypotheses, where in each round it proposes a hypothesis $\lambda(S)$ for the current sample $S \in \mathcal{S}$, and then adds the feedback $\tau(\lambda(S))$ of the teacher to obtain the new sample.

Definition 5. Let (A, \mathcal{T}) be an ALF instance with $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s), \gamma, \kappa)$, and $\mathcal{T} \subseteq \mathcal{C}$. Let $\lambda : \mathcal{S} \rightarrow \mathcal{H}$ be a learner, and let $\tau : \mathcal{H} \rightarrow \mathcal{S}$ be a teacher. The combined behavior of the learner λ and teacher τ is the function $f_{\tau, \lambda} : \mathcal{S} \rightarrow \mathcal{S}$, where $f_{\tau, \lambda}(S) := S \sqcup \tau(\lambda(S))$.

The sequence of hypotheses generated by the learner λ and teacher τ is the transfinite sequence $\langle S_{\tau, \lambda}^\alpha \mid \alpha \in \mathbb{O} \rangle$, where \mathbb{O} denotes the class of all ordinals, obtained by iterative application of $f_{\tau, \lambda}$:

- $S_{\tau, \lambda}^0 := \perp_s$;
- $S_{\tau, \lambda}^{\alpha+1} := f_{\tau, \lambda}(S_{\tau, \lambda}^\alpha)$ for successor ordinals; and

⁴ For general ALFs one has to require that the least upper bound of all samples returned by the teacher is consistent with all targets (and for non-complete sample lattices the least upper bound of all possible finite sets of samples returned by the teacher).

– $S_{\tau,\lambda}^\alpha := \bigsqcup_{\beta < \alpha} S_{\tau,\lambda}^\beta$ for limit ordinals.

If the sample lattice is not complete, the above definition is restricted to the first two items and yields a sequence indexed by natural numbers.

The following lemma states that the teacher’s properties of progress and honesty transfer to the iterative setting for consistent learners if the target specification is realizable.

Lemma 1. *Let \mathcal{T} be realizable, λ be a consistent learner, and τ be a teacher. If S is a complete sample lattice, then*

- (a) *the learner makes progress: for all $\alpha \in \mathbb{O}$, either $\kappa(S_{\tau,\lambda}^\alpha) \supsetneq \kappa(S_{\tau,\lambda}^{\alpha+1})$ and $\lambda(S_{\tau,\lambda}^\alpha) \notin \kappa(S_{\tau,\lambda}^{\alpha+1})$, or $\lambda(S_{\tau,\lambda}^\alpha) \in \mathcal{T}$, and*
- (b) *the sample sequence is consistent with the target specification: $\mathcal{T} \subseteq \kappa(S_{\tau,\lambda}^\alpha)$ for all $\alpha \in \mathbb{O}$.*

If S is a non-complete sample lattice, then (a) and (b) hold for all $\alpha \in \mathbb{N}$.

Proof. The proof is a straight-forward transfinite induction, using the properties of the teacher and the consistency relation. For the case of non-complete sample lattice, ignore the limit step in the proof below.

For part (b), the induction base is given by $\kappa(\perp_s) = \mathcal{C}$. The induction step for limit ordinals directly follows from the property of the consistency relation: all previous samples are consistent with the target specification, so their join is, too.⁵ For a successor ordinal $\alpha + 1$ it follows from the fact that $S_{\tau,\lambda}^{\alpha+1}$ is a join of two samples that are both consistent with the target specification.

For part (a), let $\alpha \in \mathbb{O}$ such that $H := \lambda(S_{\tau,\lambda}^\alpha) \notin \mathcal{T}$. Then $S_{\tau,\lambda}^{\alpha+1} = S_{\tau,\lambda}^\alpha \cup S$ with $S = \tau(H)$. In particular, $S_{\tau,\lambda}^\alpha \subseteq_s S_{\tau,\lambda}^{\alpha+1}$ and thus $\kappa(S_{\tau,\lambda}^\alpha) \supseteq \kappa(S_{\tau,\lambda}^{\alpha+1})$ by Remark 1. For the strictness of the inclusion, note that $\gamma(H) \in \kappa(S_{\tau,\lambda}^\alpha)$ because λ is a consistent learner (and $S_{\tau,\lambda}^\alpha$ is realizable because it is consistent with the realizable target specification by (b)). Furthermore, $\gamma(H) \notin \kappa(S)$ by the progress property of the teacher, and hence $\gamma(H) \notin \kappa(S_{\tau,\lambda}^{\alpha+1})$. \square

We end with an example of an ALF. Consider the problem of synthesizing guarded affine functions that capture how a piece of code P behaves, as in program deobfuscation. Then the concept class could be all functions from \mathbb{Z}^n to \mathbb{Z} , the hypothesis space would be the set of all expressions describing a guarded affine function (in some fixed syntax). The target set (as a subset of \mathcal{C}) would consist of a *single* function $\{f_t\}$, where f_t is the function computed by the program P . For any hypothesis function h , let us assume we can build a teacher who can compare h and P for equivalence, and, if they differ, return a counterexample of the form (i, o) , which is a concrete input i on which h differs from P , and o is the output of P on i . Then the sample space would consist of sets of such pairs (with union for join and empty set for \perp_s). The set of functions consistent with a

⁵ For general ALFs one uses the property that the least upper bound of all samples returned by the teacher is consistent with all targets.

set of samples would be the those that map the inputs mentioned in the samples to their appropriate outputs. The iterative learning will then model the process of synthesis, using learning, a guarded affine function that is equivalent to P .

3 Convergence of iterative learning

In this section, we study convergence of the iterative learning process. We start with a general theorem on transfinite convergence (convergence in the limit) for complete sample lattices. We then turn to convergence in finite time and exhibit three recipes that guarantee convergence.

From Lemma 1 one can conclude that the transfinite sequence of hypotheses constructed by the learner converges to a target set.

Theorem 1. *Let \mathcal{S} be a complete sample lattice, \mathcal{T} be realizable, λ be a consistent learner, and τ be a teacher. Then there exists an ordinal α such that $\lambda(S_{\tau,\lambda}^\alpha) \in \mathcal{T}$.*

Proof. Let α be an ordinal with cardinality bigger than $|\mathcal{H}|$ (bigger than $|\mathcal{S}|$ also works). If $\lambda(S_{\tau,\lambda}^\beta) \notin \mathcal{T}$ for all $\beta < \alpha$, then Lemma 1 (a) implies that all $\lambda(S_{\tau,\lambda}^\beta)$ for $\beta < \alpha$ are pairwise different, which contradicts the cardinality assumption. \square

The above theorem ratifies the choice of our definitions, and the proof (relying on Lemma 1) crucially uses all aspects of our definitions (the honesty and progress properties of the teacher, the condition imposed on κ in an ALF, the notion of consistent learners, etc.).

Convergence in finite time is clearly the more desirable notion, and we propose tactics for designing learners that converge in finite time. For an ALF instance $(\mathcal{A}, \mathcal{T})$, we say that a learner λ *converges for a teacher* τ if there is an $n \in \mathbb{N}$ such that $\lambda(S_{\tau,\lambda}^n) \in \mathcal{T}$, which means that λ produces a target hypothesis after n steps. We say that λ *converges* if it converges for every teacher. We say that λ *converges from a sample* S in case the learning process starts from a sample $S \neq \perp_s$ (i.e., if $S_{\lambda,\tau}^0 = S$).

Finite hypothesis spaces We first note that if the hypothesis space (or the concept space) is finite, then any consistent learner converges: by Lemma 1, the learner always makes progress, and hence never proposes two hypotheses that correspond to the same concept. Consequently, the learner only produces a finite number of hypotheses before finding one that is in the target (or declare that no such hypothesis exists).

There are several synthesis engines using learning that use finite hypothesis spaces. For example, Houdini [18] is a learner of *conjunctions* over a fixed finite set of predicates and, hence, has a finite hypothesis space. Learning decision trees over purely Boolean attributes (not numerical) [45] is also convergent because of finite hypothesis spaces, and this extends to the ICE learning model as well [22]. Invariant generation for arrays and lists using *elastic QDAs* [20] also uses a convergence argument that relies on a finite hypothesis space.

Occam Learners We now discuss the most robust strategy we know for convergence, based on the Occam’s razor principle. Occam’s razor advocates parsimony or simplicity [7], that the simplest concept/theory that explains a set of observations is better, as a virtue in itself. There are several learning algorithms that use parsimony as a learning bias in machine learning (e.g., *pruning* in decision-tree learning [38]), though the general applicability of Occam’s razor in machine learning as a sound means to generalize is debatable [17]. We now show that in *iterative* learning, following Occam’s principle leads to convergence in finite time. However, the role of *simplicity* itself is not the technical reason for convergence, but that there is *some* ordering of concepts that biases the learning.

Enumerative learners are a good example of this. In enumerative learning, the learner enumerates hypotheses in some order, and always conjectures the first consistent hypothesis. In an iterative learning-based synthesis setting, such a learner always converges on some target concept, if one exists, in finite time.

Requiring a total order of the hypotheses is in some situations too strict. If, for example, the hypothesis space consists of deterministic finite automata (DFAs), we could build a learner that always produces a DFA with the smallest possible number of states that is consistent with the given sample. However, the relation \preceq that compares DFAs w.r.t. their number of states is not an ordering because there are different DFAs with the same number of states.

In order to capture such situations, we work with a *total quasi-order* \preceq on \mathcal{H} instead of a total order. A quasi-order (also called preorder) is a transitive and reflexive relation. The relation being total means that $H \preceq H'$ or $H' \preceq H$ for all $H, H' \in \mathcal{H}$. The difference to an order relation is that $H \preceq H'$ and $H' \preceq H$ can hold in a quasi-order, even if $H \neq H'$.

In analogy to enumerations, we require that each hypothesis has only finitely many hypotheses “before” it w.r.t. \preceq , as expressed in the following definition.

Definition 6. A complexity ordering is a total quasi-order \preceq such that for each $x \in \mathcal{H}$ the set $\{y \in \mathcal{H} \mid y \preceq x\}$ is finite.

The example of comparing DFAs with respect to their number of states is such a complexity ordering.

Definition 7. A consistent learner that always constructs a smallest hypothesis with respect to a complexity ordering \preceq on \mathcal{H} is called an \preceq -Occam learner.

Example 1. Consider $\mathcal{H} = \mathcal{C}$ to be the interval domain over the integers consisting of all intervals of the form $[l, r]$, where $l, r \in \mathbb{Z} \cup \{-\infty, \infty\}$ and $l \leq r$. We define $[l, r] \preceq [l', r']$ if either $[l, r] = [-\infty, \infty]$ or $\max\{|x| \mid x \in [l, r] \cap \mathbb{Z}\} \leq \max\{|x| \mid x \in [l', r'] \cap \mathbb{Z}\}$. For example, $[-4, \infty] \preceq [1, 7]$ because $4 \leq 7$. This ordering \preceq satisfies the property that for each interval $[l, r]$ the set $\{[l', r'] \mid [l', r'] \preceq [l, r]\}$ is finite (because there are only finitely many intervals using integer constants with a bounded absolute value). A standard positive/negative sample $S = (P, N)$ with $P, N \subseteq \mathbb{N}$ is consistent with all intervals that contain the elements from P and do not contain an element from N . A learner that maps S to an interval that uses integers with the smallest possible absolute value (while being consistent with

S) is an \preceq -Occam learner. For example, such a learner would map the sample $(P = \{-2, 5\}, N = \{-8\})$ to the interval $[-2, \infty]$. \square

The next theorem shows that \preceq -Occam learners ensure convergence in finite time.

Theorem 2. *If \mathcal{T} is realizable and λ is a \preceq -Occam learner, then λ converges. Furthermore, the learner converges to a \preceq -minimal target element.*

Proof. Pick any target element $T \in \mathcal{H}$, which exists because \mathcal{T} is realizable. Since τ is honest, $T \in \kappa(S_{\tau, \lambda}^n)$ for all n by Lemma 1(b). Thus, on the iterated sample sequence, a \preceq -Occam learner never constructs an element which is strictly above T w.r.t. \preceq . Since there are only finitely many hypothesis that are not strictly above T , and since the learner always makes progress according to Lemma 1, it converges to a target element in finitely many steps, which itself does not have any other target elements below, and thus is \preceq -minimal. \square

There are several existing algorithms in the literature that use such orderings to ensure convergence. Several enumeration-based solvers are convergent because of the ordering of enumeration (e.g., the generic enumerative solver for SyGuS problems [2,1]). The invariant-generation ranging over conditional linear arithmetic expressions described in [21] ensures convergence using a total quasi-order based on the number of conditionals and the values of the coefficients. The learner uses templates to restrict the number of conditionals and a constraint-solver to find small coefficients for linear constraints.

Convergence using Tractable Well Founded Quasi-Orders The third strategy for convergence in finite time that we propose is based on well founded quasi-orders, or simply well-quasi-orders. Interestingly, we know of no existing learning algorithms in the literature that uses this recipe for convergence (a technique of similar flavor is used in [11]). We exhibit in this section a learning algorithm for intervals and for conjunctions of inequalities of numerical attributes based on this recipe. A salient feature of this recipe is that the convergence actually uses the samples returned by the teacher in order to converge (the first two recipes articulated above, on the other hand, would even guarantee convergence if the teacher just replies yes/no when asked whether the hypothesis is in the target set).

A binary relation \preceq over some set X is a well-quasi-order if it is transitive and reflexive, and for each infinite sequence x_0, x_1, x_2, \dots there are indices $i < j$ such that $x_i \preceq x_j$. In other words, there are no infinite descending chains and no infinite anti-chains for \preceq .

Definition 8. *Let $(\mathcal{A}, \mathcal{T})$ be an ALF instance with $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_s, \sqcup, \perp_s), \gamma, \kappa)$. A subset of hypotheses $\mathcal{W} \subseteq \mathcal{H}$ is called wqo-tractable if*

- (a) *there is a well-quasi-order $\preceq_{\mathcal{W}}$ on \mathcal{W} , and*
- (b) *for each realizable sample $S \in \mathcal{S}$ with $\kappa_{\mathcal{H}}(S) \subseteq \mathcal{W}$, there is some $\preceq_{\mathcal{W}}$ -maximal hypothesis in \mathcal{W} that is consistent with S .*

Example 2. Consider again the example of intervals over $\mathbb{Z} \cup \{-\infty, \infty\}$ with samples of the form $S = (P, N)$ (see Example 1). Let $p \in \mathbb{Z}$ be a point and let \mathcal{I}_p be the set of all intervals that contain the point p . Then, \mathcal{I}_p is wqo-tractable with the standard inclusion relation for intervals, defined by $[\ell, r] \subseteq [\ell', r']$ iff $\ell \geq \ell'$ and $r \leq r'$. Restricted to intervals that contain the point p , this is the product of two well-founded orders on the sets $\{x \in \mathbb{Z} \mid x \leq p\}$ and $\{x \in \mathbb{Z} \mid x \geq p\}$, and as such is itself well-founded [25, Theorem 2.3]. Furthermore, for each realizable sample (P, N) , there is a unique maximal interval over $\mathbb{Z} \cup \{-\infty, \infty\}$ that contains P and excludes N . Hence, the two conditions of wqo-tractability are satisfied. (Note that this ordering on the set of *all* intervals is not a well-quasi-order; the sequence $[-\infty, 0], [-\infty, -1], [-\infty, -2], \dots$ witnesses this.) \sqcup

On a wqo-tractable $\mathcal{W} \subseteq \mathcal{H}$ a learner can ensure convergence by always proposing a maximal consistent hypothesis, as stated in the following lemma.

Lemma 2. *Let \mathcal{T} be realizable, $\mathcal{W} \subseteq \mathcal{H}$ be wqo-tractable with well-quasi-order $\preceq_{\mathcal{W}}$, and S be a sample such that $\kappa_{\mathcal{H}}(S) \subseteq \mathcal{W}$. Then, there exists a learner that converges from the sample S .*

Proof. For any sample S' with $S \sqsubseteq_s S'$, the set $\kappa_{\mathcal{H}}(S')$ of hypotheses consistent with S' is a subset of \mathcal{W} . Therefore, there is some $\preceq_{\mathcal{W}}$ -maximal element in \mathcal{W} that is consistent with S' . The strategy of the learner is to return such a maximal hypothesis. Assume, for the sake of contradiction, that such a learner does not converge from S for some teacher τ . Let H_0, H_1, \dots be the infinite sequence of hypothesis produced by λ and τ starting from S , and let S_0, S_1, S_2, \dots be the corresponding sequence of samples (with $S_0 = S$). The well-foundedness of $\preceq_{\mathcal{W}}$ implies that there are $i < j$ with $H_i \preceq_{\mathcal{W}} H_j$. However, $S_i \sqsubseteq_s S_j$ because S_j is obtained from S_i by joining answers of the teacher. Therefore, H_j is also consistent with S_i (Remark 1). This contradicts the choice of H_i as a maximal hypothesis that is consistent with S_i . \square

As shown in Example 2, for each $p \in \mathbb{Z}$, the set \mathcal{I}_p of intervals containing p is wqo-tractable. Using this, we can build a convergent learner starting from the empty sample \perp_s . First, the learner starts by proposing the empty interval, the teacher must either confirm that this is a target or return a positive example, that is, a point p that is contained in every target interval. Hence, the set of hypotheses consistent with this sample is wqo-tractable and the learner can converge from here on as stated in Lemma 2. In general, the strategy for the learner is to force in one step a sample S such that the set $\kappa_{\mathcal{H}}(S) = \mathcal{I}_p$ is wqo-tractable. This is generalized in the following definition.

Definition 9. *We say that an ALF is wqo-tractable if there is a finite set $\{H_1, \dots, H_n\}$ of hypotheses such that $\kappa_{\mathcal{H}}(S)$ is wqo-tractable for all samples S that are inconsistent with all H_i , that is, $\kappa_{\mathcal{H}}(S) \cap \{H_1, \dots, H_n\} = \emptyset$.*

As explained above, the interval ALF is wqo-tractable with the set $\{H_1, \dots, H_n\}$ consisting only of the empty interval.

Combining all the previous observations, we obtain convergence for wqo-tractable ALFs.

Theorem 3. *For every ALF instance $(\mathcal{A}, \mathcal{T})$ such that \mathcal{A} is wqo-tractable and \mathcal{T} is realizable, there is a convergent learner.*

Proof. A convergent learner can be built as follows. Let $\{H_1, \dots, H_n\}$ be the finite set of hypotheses from the definition of wqo-tractability of an ALF.

- As long as the current sample is consistent with some H_i , propose such an H_i .
- Otherwise, the current sample S is such that $\kappa_{\mathcal{H}}(S)$ is wqo-tractable, and thus the learner can apply the strategy from Lemma 2. \square

A convergent learner for conjunctive linear inequality constraints. We have illustrated wqo-tractability for intervals in Example 2. We finish this section by showing that this generalizes to higher dimensions, that is, to the domain of n -dimensional hyperrectangles in $(\mathbb{Z} \cup \{-\infty, \infty\})^n$, which form the hypothesis space in this example. Each such hyperrectangle is a product of intervals over $(\mathbb{Z} \cup \{-\infty, \infty\})^n$. Note that hyperrectangles can, e.g., be used to model conjunctive linear inequality constraints over a set $f_1, \dots, f_n : \mathbb{Z}^d \rightarrow \mathbb{Z}$ of numerical attributes.

The sample space depends on the type of target specification that we are interested in. We consider here the typical sample space of positive and negative samples (however, the reasoning below also works for other sample spaces, e.g., ICE sample spaces that additionally include implications). So, samples are of the form $S = (P, N)$, where P, N are sets of points in \mathbb{Z}^n interpreted as positive and negative examples (as for intervals, see Example 1).

The following lemma provides the ingredients for building a convergent learner based on wqo-tractability.

Lemma 3. (a) *For each realizable sample $S = (P, N)$, there are maximal hyperrectangles that are consistent with S (possibly more than one).*
(b) *For each $p \in \mathbb{Z}^n$, the set \mathcal{R}_p of hyperrectangles containing p is well-quasi-ordered by inclusion.*

Proof. For the first claim, note that for each increasing chain $R_0 \subseteq R_1 \subseteq \dots$ of hyperrectangles that are all consistent with S , the union $R := \bigcup_{i \geq 0} R_i$ is also a hyperrectangle that is consistent with S . More precisely, if $R_i = [l_1^i, r_1^i] \times \dots \times [l_n^i, r_n^i]$, then $R = [l_1, r_1] \times \dots \times [l_n, r_n]$ with $l_j = \inf\{l_j^i \mid i \geq 0\}$ and $r_j = \sup\{r_j^i \mid i \geq 0\}$.

Furthermore, if the chain is strictly increasing, then there exists j such that $l_j = -\infty$ and all $l_j^i \neq -\infty$, or $r_j = \infty$ and all $r_j^i \neq \infty$. Hence, if R itself can be extended again into an infinite strictly increasing chain, then the union of this chain will contain an additional ∞ or $-\infty$. This can happen at most $2n$ times before reaching the hyperrectangle containing all points, which is certainly maximal. Thus, there has to be a maximal hyperrectangle consistent with S .

We now prove the second claim. For a point $p = (p_1, \dots, p_n)$, the set \mathcal{R}_p is the product $\mathcal{R}_p = \mathcal{I}_{p_1} \times \dots \times \mathcal{I}_{p_n}$ of the sets of intervals containing the points p_i . Furthermore, the inclusion order for hyperrectangles is the n -fold product of the inclusion order for intervals. Thus, the inclusion order on \mathcal{R}_p is a well-quasi-order because it is a product of well-quasi-orders [25]. \square

We conclude that the following type of learner is convergent: for the empty sample, propose the empty hyperrectangle; for every non-empty sample S , propose a maximal hyperrectangle consistent with S .

4 Synthesis Problems Modeled as ALFs

In this section, we list a host of existing synthesis problems and algorithms that can be seen as ALFs. Specifically, we consider examples from the areas of program verification and program synthesis. We encourage the reader to look up the referenced algorithms to better understand their mapping into our framework. Moreover, we have new techniques based on ALFs to compute fixed-points in the setting of abstract interpretation using learning.

4.1 Program Verification

While program verification itself does not directly relate to synthesis, most program verification techniques require some form of help from the programmer before the analysis can be automated. Consequently, synthesizing objects that replace manual help has been an area of active research. We here focus on *learning loop invariants*. Given adequate invariants (in terms of pre/post-conditions, loop invariants, etc.), the rest of the verification process can often be completely automated [19,26] using logical constraint-solvers [39,9]. For the purposes of this article, let us consider while-programs with a single loop. Given a pre- and post-condition, assertions, and contracts for functions called, the problem is to find a loop invariant that proves the post-condition and assertions (assuming the program is correct).

Invariant synthesis using the ICE learning model Given a program with a single loop whose loop invariant we want to synthesize, there are *many* inductive invariants that prove the assertions in the program correct— these invariants are characterized by the following three properties: (a) that it include the states when the loop is entered the first time, (b) that it exclude the states that immediately exit the loop and reach the end of the program and not satisfy the post-condition, and (c) that it is inductive (i.e., from any state satisfying the invariant, if we execute the loop body once, the resulting state is also in the invariant). The teacher knows these properties, and must reply to conjectured hypotheses of the learner using these properties. Violation of properties (a) and (b) are usually easy to check using a constraint solver, and will result in a *positive* and *negative* concrete configuration as a sample, respectively. However, when inductiveness fails, the obvious counterexample is a *pair* of configurations, (x, y) , where x is in the hypothesis but y is not, and where the program state x evolves to the state y across one execution of the loop body.

The work by Garg et. al. [21] hence proposes what they call the *ICE model* (for implication counterexamples), where the learner learns from positive, negative, and implication counterexamples. The author’s claim is that without implication

counterexamples, the teacher is stuck when presented a hypothesis that satisfies the properties of being an invariant save the inductiveness property.

From the described components we build an ALF $\mathcal{A}_{\text{ICE}} = (\mathcal{C}, \mathcal{H}, \gamma, \mathcal{S}, \kappa)$, where \mathcal{C} is the set of all subsets of program configurations, the hypothesis space \mathcal{H} is the language used to describe the invariant, and the sample space is defined as follows:

- A sample is of the form $S = (P, N, I)$, where P, N are sets of program configurations (interpreted as positive and negative examples), and I is a set of pairs of program configurations (interpreted as implications).
- A set $C \in \mathcal{C}$ of program configurations is consistent with (P, N, I) if $P \subseteq C$, $N \cap C = \emptyset$, and if $(c, c') \in I$ and $c \in C$, then also $c' \in C$.
- The order on samples is defined by component-wise set inclusion (i.e., $(P, N, I) \sqsubseteq_s (P', N', I')$ if $P \subseteq P'$, $N \subseteq N'$, and $I \subseteq I'$).
- The join is the component-wise union, and $\perp_s = (\emptyset, \emptyset, \emptyset)$.

Since this sample space contains implications in addition to the standard positive and negative examples, we refer to it as an *ICE sample space*.

We can now show that there is a teacher for these ALF instances, because a teacher can refute any hypothesis made by a learner with a positive, negative, or implication counterexample, depending on which property of invariants is violated.

Proposition 1. *There is a teacher for ALF instances of the form $(\mathcal{A}_{\text{ICE}}, \mathcal{T}_{\text{Inv}})$.*

Furthermore, we can show that having only positive and negative samples precludes the existence of teachers. In fact, we can show that if $\mathcal{C} = 2^D$ (for a domain D) and the sample space \mathcal{S} consists of only positive and negative examples in D , then a target set \mathcal{T} has a teacher only if it is defined in terms of excluding a set B and including a set G .

Lemma 4. *Let $C = H = 2^D$, $\gamma = \text{id}$, $S = \{(P, N) \mid P, N \subseteq D\}$, and $\kappa((P, N)) = \{R \subseteq D \mid P \subseteq R \wedge R \cap N = \emptyset\}$.*

Let $\mathcal{T} \subseteq \mathcal{C}$ be a target. If there exists a teacher for \mathcal{T} , then there must exist sets $B, G \subseteq D$ such that $\mathcal{T} = \{R \subseteq D \mid B \cap R = \emptyset \text{ and } G \subseteq R\}$.

Proof. Assume that there exists a teacher for the target set \mathcal{T} , and let G and B be the union of all positive examples and the union of all negative examples, respectively, that the teacher returns. Now, we claim that $\mathcal{T} = \{R \subseteq D \mid B \cap R = \emptyset \text{ and } R \subseteq G\}$. Towards a contradiction, assume that this is not the case. Then, there exists an $R \in \mathcal{T}$ such that $B \cap R \neq \emptyset$ or $G \not\subseteq R$. If $B \cap R \neq \emptyset$, then there is some $b \in R$ that was returned as a negative counterexample for some hypothesis. Since $R \in \mathcal{T}$ is not consistent with this negative example b , this contradicts the requirement that the teacher is honest. Similarly, if $G \not\subseteq R$, then there is some $g \in G$ that was returned as a positive counterexample, which contradicts the teacher's honesty.

The above proves that positive and negative samples are not sufficient for learning invariants, as invariants cannot be defined as all sets that exclude a set of states and include a set of states.

There are several ICE-based invariant synthesis formalisms that we can capture. First, Garg et. al. [21] have considered arithmetic invariants over a set of integer variables x_1, \dots, x_ℓ of the form $\bigvee_{i=1}^n \bigwedge_{j=1}^{m_i} (\sum_{k=1}^\ell a_k^{i,j} x_k \leq c^{i,j})$, $a_k^{i,j} \in \{-1, 0, 1\}$, where the learner is implemented using a constraint solver that finds smallest invariants that fit the sample. This is accurately modeled in our framework, as in the ICE formulation above, with the hypotheses space being the set of all formulas of this form. The fact that Garg et. al.’s learner produces smallest invariants makes it an Occam learner in the sense of Section 3 and, hence, it converges in finite time. The approach proposed by Sharma and Aiken [49], C2I, is also an ICE-learner, except that the learner uses *stochastic search* based on a Metropolis Hastings MCMC (Markov chain Monte Carlo) algorithm, which again can be seen as an ALF.

We can also see the work by Garg et. al. [20] on synthesizing quantified invariants for linear data structures such as lists and arrays as ALFs. This framework can infer quantified invariants of the form

$$\forall y_1, y_2: (y_1 \leq y_2 \leq i) \Rightarrow a[y_1] \leq a[y_2].$$

However, Garg et. al. do not represent sets of configurations by means of logical formulas (as shown above) but use an automata-theoretic approach, where a special class of automata, called *quantified data automata (QDAs)*, represent such logical invariants; hence, these QDAs form the hypothesis space in the ALF. The sample space there is also unusual: a sample (modeling a program configuration consisting of arrays or lists) is a *set of valuation words*, where each such word encodes the information about the array (or list) for specially quantified pointer variables pointing into the heap, and where data-formulas state conditions of the keys stored at these locations.

ALFs can also capture the ICE-framework described by Neider [40], where invariants are learned in the context of *regular model-checking* [12]. In regular model checking, program configurations are captured using (finite—but unbounded—or even infinite) words, and sets of configurations are captured using finite automata. Consequently, the hypothesis space is the set of all DFAs (over an a priori chosen, fixed alphabet), and the sample space is an ICE-sample consisting of configurations modeled as words. The learner proposed by Neider constructs consistent DFAs of minimal size and, hence, is an Occam learner that converges in finite time (cf. Section 3).

We now turn to two other invariant-generation frameworks that skirt the ICE model.

Houdini The Houdini algorithm [18] is a learning algorithm for synthesizing invariants that avoids learning from ICE samples. Given a finite set of *predicates*

P , Houdini learns an invariant that is expressible as a conjunction of some subset of predicates (note that the hypothesis space is finite but exponential in the number of predicates). Houdini learns an invariant in time *polynomial* in the number of predicates (and in linear number rounds) and is implemented in the Boogie program verifier [8]. It is widely used (for example, used in verifying device drivers [35,36] and in race-detection in GPU kernels [10]).

The setup here can be modeled as an ALF: we take the concept space \mathcal{C} to be all subsets of program configurations, and the hypothesis space \mathcal{H} to be the set of all conjunctions of subsets of predicates in P , with the map γ mapping each conjunctive formula in \mathcal{H} to the set of all configurations that satisfy the predicates mentioned in the conjunction. We take the sample space to be the ICE sample space, where each sample is a valuation v over P (indicating which predicates are satisfied) and where implication counterexamples are pairs of valuations.

The Houdini learning algorithm itself is the classical conjunctive learning algorithm for positive and negative samples (see [30]), but its mechanics are such that it works for ICE samples as well. More precisely, the Houdini algorithm always creates the *semantically smallest* formula that satisfies the sample (it hence starts with a conjunction of all predicates, and in each round “knocks off” predicates that are violated by positive samples returned). Since Houdini always returns the semantically-smallest conjunction of predicates, it will never receive a negative counterexample (assuming the program is correct and has a conjunctive invariant over P). Furthermore, for an implication counterexample (v, v') , the algorithm knows that since it proposed the semantically smallest conjunction of predicates, v cannot be made negative; hence it treats v' as a positive counterexample. Houdini converges since the hypothesis space is finite (matching the first recipe for convergence we outlined in Section 3); in fact, it converges in linear number of rounds since in each round at least one predicate is removed from the hypothesized invariant.

Learning invariants in regular model-checking using witnesses The learning-to-verify project reported in [56,55,57,58] leverages machine learning to the verification of infinite state systems that result from processes with FIFO queues, but skirts the ICE model using a different idea. The key idea is to view the identification of the reachable states of such a system as a machine learning problem instead of computing this set iteratively (which, in general, requires techniques such as acceleration or widening). In particular, we consider the work of Vardhan et. al. [56] and show that this is an instantiation of our abstract learning framework. The key idea in this work is to represent configurations as traces through the system and to add a notion of *witness* to this description, resulting in so-called annotated traces. The teacher, when receiving a set of annotated traces, can actually check whether the configurations are reachable based on the witnesses (a witness can be, say, the length of the execution or the execution itself) and, consequently, the model allows learning from such traces directly. Indeed, this approach can be modeled by an ALF: the concept space consists of subsets of configurations, the target space consists of the set of reachable configurations,

the hypothesis space consists of automata over annotated traces, and the sample space consists of positive and negatively labeled annotated traces.

4.2 Synthesis of Fixpoints in Abstract Domains

In Section 4.1 we have explained how several instances of ICE learners fit into our framework. We now provide a generic technique to model the problem of fixpoint computation in the setting of abstract interpretation using learning.

We assume the setting of abstract interpretation [16] with

- a concrete domain $(\mathcal{D}, \subseteq, \perp, \cup)$ and an abstract domain $(\widehat{\mathcal{D}}, \sqsubseteq, \perp, \sqcup)$, which both have a join lattice structure, and
- a Galois connection between these two, given by two monotone functions $\gamma : \widehat{\mathcal{D}} \rightarrow \mathcal{D}$ (the concretization function), and $\alpha : \mathcal{D} \rightarrow \widehat{\mathcal{D}}$ (the abstraction function), with $\alpha(X) \sqsubseteq \widehat{X} \Leftrightarrow X \subseteq \gamma(\widehat{X})$ for all $X \in \mathcal{D}$ and $\widehat{X} \in \widehat{\mathcal{D}}$.

The concrete domain usually describes the semantics of a program and comes with an increasing transformer $F : \mathcal{D} \rightarrow \mathcal{D}$ that captures the behaviour of the program (increasing means that $X \subseteq F(X)$ for each X). The abstract domain is used to model the aspects of the program that one is interested in.

A specification is given in terms a set $B \subseteq \mathcal{D}$ of bad concrete elements. The goal is to find a fixpoint of F that is not above any bad element, i.e., an element $X \in \mathcal{D}$ with $F(X) = X$, and $Y \not\subseteq X$ for each $Y \in B$. We refer to such fixpoints as *adequate fixpoints* because they show that the program cannot reach a bad state.

Synthesis of Precise Abstract Transformers The general idea of abstract interpretation is to do the fixpoint computation on the abstract domain instead of the concrete domain. This is done using an abstract transformer $\widehat{F} : \widehat{\mathcal{D}} \rightarrow \widehat{\mathcal{D}}$ that overapproximates the concrete transformer, in the sense that $\alpha(F(\gamma(\widehat{X}))) \sqsubseteq \widehat{F}(\widehat{X})$ for each $\widehat{X} \in \widehat{\mathcal{D}}$. The best abstract transformer is the one where equality holds instead of inclusion.

Since manually designing a good abstract transformer can be difficult, Thakur et al. [53] propose an automatic method to synthesize the abstract post of a given abstract element for the concrete domain consisting of sets of program configurations $\mathcal{D} = 2^D$.

We can model the algorithm proposed by Thakur et al. [53] as a synthesis using learning in an ALF that uses the concrete domain \mathcal{D} as concept space, the abstract domain $\widehat{\mathcal{D}}$ as hypothesis space together with the existing concretization function γ as the concretization function for the ALF. Since we are interested in elements above $\alpha(F(\gamma(\widehat{X})))$, the sample space simply consists of positive examples, that is, a sample is a finite set of elements of D . A hypothesis is consistent with a sample if it contains all the elements from the sample.

For an abstract element \widehat{X} , the target specification is given by $\mathcal{T} = \{T \subseteq D \mid \alpha(F(\gamma(\widehat{X}))) \subseteq T\}$.

If the learner proposes some hypothesis H (which is an abstract element), then the teacher can check whether $\gamma(H)$ is a superset of $F(\gamma(\hat{X}))$ and if not, return a positive example from $F(\gamma(\hat{X})) \setminus \gamma(H)$. This is precisely what happens (at an abstract level) in the procedure proposed in [53].

One should note here that in case the abstract domain has a maximal element, a learner could always propose this element because it can be sure that it is a target element if the target is realizable. The idea is to come up with a learner that computes a better solution than just the maximum. In [53] this is done by starting with the empty hypothesis, and then for each positive example S returned by the teacher, applying the abstraction function α on it and taking the join of the resulting abstract element with the current hypothesis. In this way, the learner ensures that the hypothesis is always below the set $\alpha(F(\gamma(\hat{X})))$. So if the learner converges, then it computes the best abstract transformer on \hat{X} .

Synthesis of Adequate Fixpoints. We now describe a generic way for designing an ALF that models the problem of finding an adequate fixpoint as a synthesis problem that uses learning from samples. This can be seen as a generalization of finding loop invariants in an abstract domain (but without computing the abstract transformers).

The ALF we propose has the concrete domain \mathcal{D} as concept space, the abstract domain $\hat{\mathcal{D}}$ as hypothesis space together with the existing concretization function γ . The sample space of an ALF is, in general, designed for a specific class of target specifications (in order to guarantee the existence of a teacher). In the following, we describe how to construct such a sample space for adequate fixpoints as targets.

The invariants defined in Section 4.1 can be seen as adequate fixpoints for the transformer F and the set B being defined pointwise on single configurations (with concrete domain consisting of sets of program configurations). This pointwise definition is the reason why the sample space can be build up by using single configurations as positive and negative examples, and pairs of configurations as implications.

We replace the pointwise definition by a definition of in terms of a class $\mathcal{R} \subseteq \mathcal{D}$ of representative concrete elements. So the definition below is satisfied by the class of singleton sets in case of the concrete domain consisting of sets of program configurations. It intuitively states that \mathcal{R} is rich enough to prove that a hypothesis is not a fixpoint.

Definition 10. *A set $\mathcal{R} \subseteq \mathcal{D}$ is a representative set (for the transformer F) if for each $\hat{X} \in \hat{\mathcal{D}}$ and $X := \gamma(\hat{X})$ such that $F(X) \neq X$, there are $Y, Y' \in \mathcal{R}$ with $Y \subseteq X$, $Y' \subseteq F(Y)$, and $Y' \not\subseteq X$. \square*

It is not hard to see that \mathcal{D} is always representative set. And in fact that the set of all elements of the form $\gamma(H)$ and $F(\gamma(H))$, where $H \in \mathcal{H}$ also forms a representative set.

We consider target specifications of adequate fixpoints that can be expressed in terms of \mathcal{R} . We say that $\mathcal{T} \subseteq \mathcal{D}$ is an \mathcal{R} -specification of adequate fixpoints if there is a set $B \subseteq \mathcal{R}$ such that $\mathcal{T} = \{X \in \mathcal{D} \mid F(X) = X \text{ and } Y \not\subseteq X \text{ for all } Y \in B\}$.

For this class of target specifications, we let the sample space $\mathcal{S}_{\mathcal{R}}$ consist of ICE samples over \mathcal{R} , that is, of triples (P, N, I) of finite set $P, N \subseteq \mathcal{R}$, and a finite set $I \subseteq \mathcal{R} \times \mathcal{R}$. A concept $X \in D$ is consistent with a sample (P, N, I) if $Y \subseteq X$ for all $Y \in P$, $Y \not\subseteq X$ for all $Y \in N$, and if $Y \subseteq X$, then $Y' \subseteq X$ for all $(Y, Y') \in I$. This consistency relation is denoted by $\kappa_{\mathcal{R}}$. In analogy to Proposition 1 we can show that this sample space is expressive enough to guarantee the existence of a teacher.

Proposition 2. *There is a teacher for ALF instances of the form $(\mathcal{A}, \mathcal{T})$ with $\mathcal{A} = (\mathcal{D}, \widehat{\mathcal{D}}, \gamma, \mathcal{S}_{\mathcal{R}}, \kappa_{\mathcal{R}})$ and an \mathcal{R} -specification of adequate fixpoints \mathcal{T} .*

This provides a generic way of setting up a learning scenario for abstract interpretation, and thus provides a powerful tool for understanding the requirements for the application and development of machine learning algorithms for the synthesis of adequate fixpoints.

4.3 Program Synthesis

In this section, we study several examples of learning-based program synthesis, which include synthesizing program expressions, expressions to be plugged in program sketches, snippets of programs, etc., and show how they can be modeled as ALFs.

End-user synthesis from examples: Flashfill One application of synthesis is to use it to help end-users to program using examples. A prime example of this is FLASHFILL by Gulwani et al [23], where the authors show how string manipulation macros from user-given input-output examples can be synthesized in the context of Microsoft Excel spreadsheets. Flashfill can be seen as an ALF: the concept space consists of all functions from strings to strings, the hypothesis space consists of all string manipulation macros, and the sample space consists of a sets of input-output examples for such functions. The consistency relation κ maps each sample to all functions that agree with the sample. The role of the teacher is played by the *user*: the user has some function in mind and gives new input-output examples whenever the learner returns a hypothesis that she is not satisfied with. The learning algorithm here is based on version-space algebras (which, intuitively, compactly represents *all* possible macros with limited size that are consistent with the sample) and in each round proposes a simple macro from this collection.

Completing sketches and the SyGuS solvers The sketch-based synthesis approach [50] is another prominent synthesis application, where programmers write partial programs with holes and a system automatically synthesizes expressions or programs for these holes so that a specification (expressed using input-output pairs or logical assertions) is satisfied. The key idea here is that given a sketch with a specification, we need expressions for the holes such that *for every*

possible input, the specification holds. This roughly has the form $\exists \mathbf{e}.\forall \mathbf{x}\psi(\mathbf{e}, \mathbf{x})$, where \mathbf{e} are the expressions to synthesize and \mathbf{x} are the inputs to the program.

The Sketch system works by (a) unfolding loops a finite number of times, hence, bounding the length of executions, and (b) encoding the choice of expressions \mathbf{e} to be synthesized using bits (typically using templates and representing integers by a small number of bits). For the synthesis step, the Sketch system implements a CEGIS (counterexample guided synthesis) technique using SAT solving, whose underlying idea is to learn the expressions from examples using only a SAT solver. The CEGIS technique works in rounds: the learner proposes hypothesis expressions and the teacher checks whether $\forall \mathbf{x}\psi(\mathbf{e}, \mathbf{x})$ holds (using SAT queries) and if not, returns a valuation for \mathbf{x} as a counterexample. Subsequently, the learner asks, again using a SAT query, whether there exists a valuation for the bits encoding the expressions such that $\psi(\mathbf{e}, \mathbf{x})$ holds for every valuation of \mathbf{x} returned by the teacher thus far; the resulting expressions are the hypotheses for the next round. Note that the use of samples avoids quantifier alternation both in the teacher and the learner.

The above system can be modeled as an ALF. The concept space consists of tuples of functions modeling the various expressions to synthesize, the hypothesis space is the set of expressions (or their bit encodings), the map γ gives meaning to these expressions (or encodings), and the sample space can be seen as the set of *grounded formulae* of the form $\psi(\mathbf{e}, \mathbf{v})$ where the variables \mathbf{x} have been substituted with a concrete valuation. The relation κ maps such a sample to the set of all expressions \mathbf{f} such that the formulas in the sample all evaluate to true if \mathbf{f} is substituted for \mathbf{e} . The Sketch learner can be seen as a learner in this ALF framework that uses calls to a SAT solver to find hypothesis expressions consistent with the sample. Since expressions are encoded by a finite number of bits, the hypothesis space is finite, and the Sketch learner converges in finite time (cf. Section 3).

The SyGuS format [2] is a competition format for synthesis, and extends the Sketch-based formalism above to SMT theories, with an emphasis on syntactic restrictions for expressions. More precisely, SyGuS specifications are parameterized over a background theory T , and an instance is a pair $(G, \psi(\mathbf{f}))$ where G is a grammar that imposes syntactic restrictions for functions (or expressions) written using symbols of the background theory, and ψ is a formula, again in the theory T , including function symbols \mathbf{f} ; the functions \mathbf{f} are typed according to domains of T . The goal is to find functions \mathbf{g} for the symbols \mathbf{f} in the syntax G such that ψ holds. The competition version further restricts ψ to be of the form $\forall \mathbf{x}\psi'(\mathbf{f}, \mathbf{x})$ where ψ' is a quantifier-free formula in a decidable SMT theory—this way, given a hypothesis for the functions \mathbf{f} , the problem of checking whether the functions meet the specification is decidable.

There have been several solvers developed for SyGuS (cf. the first SyGuS competition [2,1]), and all of them are in fact learning-based (i.e., CEGIS) techniques. In particular, three solvers have been proposed: an enumerative solver, a constraint-based solver, and a stochastic solver. All these solvers can be seen as ALF instances: the concept space consists of all possible tuples of functions

over the appropriate domains and the hypothesis space is the set of all functions allowed by the *syntax* of the problem (with the natural γ relation giving its semantics). All three solvers work by generating a tuple of functions such that $\forall \mathbf{x} \psi'(\mathbf{f}, \mathbf{x})$ holds for all valuations of \mathbf{x} given by the teacher thus far. The enumerative solver enumerates functions until it reaches such a function, the stochastic solver searches the space of functions randomly using a measure that depends on how many samples are satisfied till it finds one that satisfies the samples, and the constraint-based solver queries a constraint-solver for instantiations of template functions so that the specification is satisfied on the sample valuations. Both the enumerative and the constraint-solver are Occam learners and, hence, converge in finite time.

Note that the learners *know* ψ in this scenario. However, we can model SyGuS as ALFs by taking the sample space to be grounded formulas $\psi'(\mathbf{f}, \mathbf{v})$ consisting of the specification with particular values \mathbf{v} substituted for \mathbf{x} . The learners can now be seen as learning from these samples, without knowledge of ψ (similar to the modeling of Sketch above).

We would like to emphasize that this embedding of SyGuS as an ALF clearly showcases the difference between different synthesis approaches (as mentioned in the introduction). For example, invariant generation can be done using learning either by means of ICE samples (see Section 4.1) or modeled as a SyGuS problem. However, it turns out that the sample spaces (and, hence, the learners) in the two approaches are *very different*! In ICE-based learning, samples are only single configurations (labeled positive or negative) or pairs of configurations, while in a SyGuS encoding, the samples are grounded formulas that encode the entire program body, including instantiations of universally quantified variables intermediate states in the execution of the loop.

Machine-learning based approaches to synthesis One can implement the *passive* machine-learning algorithm to synthesize hypotheses from samples, in order to build a synthesis engine (along with an appropriate teacher that can furnish such samples). Recent work by Garg et. al. [22] proposes an algorithm for synthesizing invariants in the ICE-framework using machine learning classifiers (decision trees) that can be viewed as an ALF.

Synthesizing guarded affine functions Recent work [47] explores the synthesis of *guarded affine functions* from a sample space that consists of information of the form $f(\mathbf{s}) = t$, where \mathbf{s} and t are integers. The learner here uses a combination of computational geometry techniques and decision tree learning, and can also be modeled as an ALF. Notice that this sample space precisely matches the sample space for deobfuscation problems (where the teacher can return counterexamples of this form – see Example at the end of Section 2 on page 8 – using the program being deobfuscated). Consequently, the learner in Alchemist [47] can be used for deobfuscating programs that compute guarded affine functions from tuples of integer inputs to integers (like the “multiply by 45” example in [27]).

Other synthesis engines There are several algorithms that are self-described as CEGIS frameworks, and, hence, can be modeled using ALFs. For example, synthesizing loop-free programs [24], synthesizing synchronizing code for concurrent programs [14] (in this work, the sample space consists of abstract concurrent partially-ordered traces), work on using synthesis to *mine specifications* [28], synthesizing bit-manipulating programs and deobfuscating programs [27] (here, the use of separate I/O-oracle can be modeled as the teacher returning the output of the program together with a counterexample input), superoptimization [48], deductive program repair [32], synthesis of recursive functional programs over unbounded domains [33], as well as synthesis of protocols using enumerative CEGIS techniques [54].

5 Variations and Limitations of the Framework

In this section we discuss some variations and limitations of our framework. We start by briefly discussing a variation of our framework that omits the concept space.

5.1 Omitting the Concept Space

We believe that, for a clean modeling of a synthesis problem, one should specify the concept space \mathcal{C} . This makes it possible to compare different synthesis approaches that work with different representations of hypotheses and maybe different types of samples over the same underlying concept space.

However, for the actual learning process, the concept space itself is not of great importance because the learner proposes elements from the hypothesis space, and the teacher returns an element from the sample space. The concept space only serves as a semantic space that gives meaning to hypotheses (via the concretization function γ), and to the samples (via the consistency relation κ).

Therefore, it is possible to omit the concept space from an ALF, and to directly specify the consistency of samples with hypotheses. Such a reduced ALF would then be of the form $\mathcal{A} = (\mathcal{H}, \mathcal{S}, \kappa)$ with a function $\kappa : \mathcal{S} \rightarrow 2^{\mathcal{H}}$. In the original framework, this corresponds to the function $\kappa_{\mathcal{H}}$ defined by $\kappa_{\mathcal{H}}(S) = \gamma^{-1}\kappa(S)$.

To create ALF instances, the target specification is also directly given as a subset of the hypothesis space $\mathcal{T} \subseteq \mathcal{H}$. All the other definitions can be adapted directly to this framework.

5.2 Limitations

The ALF framework we develop in this paper is not meant to capture every existing method that uses learning from samples. There are several synthesis techniques that use grey-box techniques (a combination of black-box learning from samples and by utilizing the specification of the target directly in some way) or use query models (where they query the teacher for various aspects of the target set).

For instance, there are active iterative learning scenarios in which the learner can ask other types of questions to the teacher than just proposing hypotheses that are then accepted or refuted by the teacher. One prominent scenario of this kind is Angluin’s active learning of DFAs [5], where the learner can ask *membership queries* and *equivalence queries*. (The equivalence queries correspond to proposing a hypothesis, as in our framework, which is then refuted with a counterexample if it is not correct.) Such learning scenarios for synthesis are used, for example, in [3] for the synthesis of interface specifications for Java classes, and in [43] for automatically synthesizing assumptions for assume-guarantee reasoning. Our framework does not have a mechanism for directly modeling such queries. The ALF framework that we have presented is intentionally a simpler framework by design that captures and cleanly models emerging synthesis procedures in the literature where the learner only proposes hypotheses and learns from samples the teacher provides in terms of samples to show that the hypothesis is wrong. The learner in our framework, being a completely passive learner (as opposed to an active learner), can also be implemented by the variety of scalable passive machine-learning algorithms in vogue [38]. A clean extension of ALFs to query settings and grey-box settings would be an interesting future direction to pursue.

6 Conclusions

We have presented an abstract learning framework for synthesis that encompasses several existing techniques that use learning or counter-example guided inductive synthesis to create objects that satisfy a specification. We were motivated by abstract interpretation [16] and how it gives a general framework and notation for verification; our formalism is an attempt at such a generalization for learning-based synthesis. The conditions we have proposed that the abstract concept spaces, hypotheses spaces, and sample spaces need to satisfy to define a learning-based synthesis domain seem to be cogent and general in forming a vocabulary for such approaches. We have also addressed various strategies for convergent synthesis that generalizes and extends existing techniques (again, in a similar vein as to how widening and narrowing in abstract interpretation give recipes for building convergent algorithms to compute fixed-points). We believe that the notation and general theorems herein will bring more clarity, understanding, and reuse of learners in synthesis algorithms.

Acknowledgements: This work was partially supported by NSF Expeditions in Computing ExCAPE Award #1138994.

References

1. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Comm. Security, vol. 40, pp. 1–25. IOS Press (2015)

2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD 2013. pp. 1–8. IEEE (2013)
3. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL 2005. pp. 98–109. ACM (2005)
4. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: Results and analysis of sygus-comp’15. Tech. rep., University of Pennsylvania (2015), http://www.cis.upenn.edu/~fisman/documents/AFSS_SYNT15.pdf
5. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* 75(2), 87–106 (1987)
6. Angluin, D.: Computational learning theory: Survey and selected bibliography. In: STOC 1992. pp. 351–369. ACM (1992)
7. Baker, A.: Simplicity. In: Zalta, E.N. (ed.) *The Stanford Encyclopedia of Philosophy*. Fall 2013 edn. (2013), <http://plato.stanford.edu/archives/fall2013/entries/simplicity/>
8. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. pp. 364–387 (2005)
9. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer (2011)
10. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: Gpuverify: a verifier for GPU kernels. In: OOPSLA 2012. pp. 113–132. ACM (2012)
11. Blum, A.: Learning boolean functions in an infinite attribute space. *Machine Learning* 9, 373–386 (1992)
12. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: CAV 2000. vol. 1855, pp. 403–418. Springer (2000)
13. Cerný, P., Clarke, E.M., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Samanta, R., Tarrach, T.: From non-preemptive to preemptive scheduling using synchronization synthesis. In: CAV 2015. LNCS, vol. 9207, pp. 180–197. Springer (2015)
14. Cerny, P., Clarke, E.M., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Samanta, R., Tarrach, T.: From non-preemptive to preemptive scheduling using synchronization synthesis (2015), CAV 2015, to appear
15. Cheung, A., Madden, S., Solar-Lezama, A., Arden, O., Myers, A.C.: Using program analysis to improve database applications. *IEEE Data Eng. Bull.* 37(1), 48–59 (2014)
16. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
17. Domingos, P.M.: The role of occam’s razor in knowledge discovery. *Data Min. Knowl. Discov.* 3(4), 409–425 (1999)
18. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME. LNCS, vol. 2021, pp. 500–517. Springer (2001)
19. Floyd, R.: Assigning meaning to programs. In: Schwartz, J.T. (ed.) *Mathematical Aspects of Computer Science*. pp. 19–32. No. 19, AMS (1967)
20. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: CAV’13. vol. 8559, pp. 813–829. Springer (2013)
21. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV 2014. vol. 8559, pp. 69–87. Springer (2014)

22. Garg, P., Madhusudan, P., Neider, D., Roth, D.: Learning invariants using decision trees and implication counterexamples. *POPL 2016* p. to appear (2016)
23. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: *POPL 2011*. pp. 317–330. ACM (2011)
24. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: *PLDI 2011*. pp. 62–73. ACM (2011)
25. Higman, G.: Ordering by Divisibility in Abstract Algebras. *Proc. London Math. Soc.* s3-2(1), 326–336 (1952)
26. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* 12(10), 576–580 (1969)
27. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: *ICSE 2010*. pp. 215–224. ACM (2010)
28. Jin, X., Donzé, A., Deshmukh, J.V., Seshia, S.A.: Mining requirements from closed-loop control models. In: *HSCC 2013*. pp. 43–52. ACM (2013)
29. Karaivanov, S., Raychev, V., Vechev, M.T.: Phrase-based statistical translation of programming languages. In: *Onward!, SLASH '14*. pp. 173–184. ACM (2014)
30. Kearns, M.J., Vazirani, U.V.: An introduction to computational learning theory. MIT Press, Cambridge, MA, USA (1994)
31. Kitzelmann, E.: Inductive programming: A survey of program synthesis techniques. In: *AAIP 2009, Revised Papers. Lecture Notes in Computer Science*, vol. 5812, pp. 50–73. Springer (2010)
32. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: *CAV 2015. LNCS*, vol. 9207, pp. 217–233. Springer (2015)
33. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: *OOPSLA 2013*. pp. 407–426. ACM (2013)
34. Kuncak, V.: Verifying and synthesizing software with recursive functions - (invited contribution). In: *ICALP 2014. LNCS*, vol. 8572, pp. 11–25. Springer (2014)
35. Lal, A., Qadeer, S.: Powering the static driver verifier using corral. In: *(FSE-22)*. pp. 202–212. ACM (2014)
36. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: *CAV 2012. LNCS*, vol. 7358, pp. 427–443. Springer (2012)
37. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.* 2(1), 90–121 (Jan 1980)
38. Mitchell, T.M.: Machine learning. McGraw-Hill (1997)
39. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: *TACAS*. pp. 337–340 (2008)
40. Neider, D.: Applications of Automata Learning in Verification and Synthesis. Ph.D. thesis, RWTH Aachen University (April 2014)
41. Neider, D., Saha, S., Madhusudan, P.: Synthesizing piece-wise functions by learning classifiers. In: *TACAS 2016*. p. to appear. LNCS, Springer (2016)
42. Osera, P., Zdanczewicz, S.: Type-and-example-directed program synthesis. In: *PLDI'15*. pp. 619–630. ACM (2015)
43. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design* 32(3), 175–205 (2008)
44. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: *POPL'89*. pp. 179–190 (1989)
45. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
46. Saha, S., Garg, P., Madhusudan, P.: Alchemist: Learning guarded affine functions. In: *CAV 2015. LNCS*, vol. 9206, pp. 440–446. Springer (2015)

47. Saha, S., Garg, P., Madhusudan, P.: Alchemist: Learning guarded affine functions (2015), CAV 2015, to appear
48. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS '13. pp. 305–316. ACM (2013)
49. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: CAV 2014. LNCS, vol. 8559, pp. 88–105. Springer (2014)
50. Solar-Lezama, A.: Program Synthesis by Sketching". Ph.D. thesis, University of California at Berkeley (2008)
51. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. pp. 404–415 (2006)
52. Thakur, A., Lal, A., Lim, J., Reps, T.: PostHat and all that: Attaining most-precise inductive invariants. Tech. Rep. TR1790, University of Wisconsin, Madison (Apr 2013)
53. Thakur, A.V., Lal, A., Lim, J., Reps, T.W.: Posthat and all that: Automating abstract interpretation. *Electr. Notes Theor. Comput. Sci.* 311, 15–32 (2015)
54. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: PLDI'13. pp. 287–296. ACM (2013)
55. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Actively learning to verify safety for FIFO automata. In: FSTTCS 2004. vol. 3328, pp. 494–505. Springer (2004)
56. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Learning to verify safety properties. In: ICFEM 2004. vol. 3308, pp. 274–289. Springer (2004)
57. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using language inference to verify omega-regular properties. In: TACAS'05. vol. 3440, pp. 45–60. Springer (2005)
58. Vardhan, A., Viswanathan, M.: Learning to verify branching time properties. In: ASE 2005. pp. 325–328. ACM (2005)